References, Local State, and Objects

Mattox Beckman

beckman@iit.edu

Illinois Institute of Technology

Objectives

The lack of mutable variables gives us the ability to perform many analyses using mathematics. In this lecture we talk about *equational reasoning* and references, and see techniques for limiting the scope of the state to improve the reliability of your code.

- Be able to explain equational reasoning and give an example.
- Know the syntax of references in OCaml.
- Know the tradeoffs between imperative and functional features.
- Know the constructions to define a function with local state.
- Be able to state the benefits of local state and give an example.
- Be able to use tuples to allow multiple functions access to the same state.

More Objectives

In this lecture we also extend the idea of local state to create a simple implementation of objects, and discuss its limitations. We will also show the message dispatch model of objects, which allows for inheritance and virtual functions.
Your objectives:

- Be able to explain what an object is.
- Know how to implement an object using records and HOFs.
- Know how to implement an object using a message dispatcher.
- Be able compare the record and dispatcher models.
- Major goal 1: be able to simulate objects in a language lacking them.
- Major goal 2: understand how objects work “under the hood”.

Definition

The rule of *referential transparency:*

$$\frac{e_1 \Downarrow v \quad e_2 \Downarrow v \quad f\ e_1 \Downarrow w}{f\ e_2 \Downarrow w}$$

- If you have two expressions that evaluate to be the same thing then you can use one for the other without changing the meaning of the whole program.
- e.g. `f(x) + f(x) == 2 * f(x)`
- You can prove this by induction, using the natural semantic rules from the previous lectures.

- You can use equational reasoning to make the following equivalence:

$$\texttt{f(if } e_1 \texttt{ then } e_2 \texttt{ else } e_3\texttt{)} \equiv \texttt{if } e_1 \texttt{ then f(}e_2\texttt{) else f(}e_3\texttt{)}$$

```
1 x * (if foo then 20 / x else 23 / x) equivalent to
2 if foo then 20 else 23   (well, mostly)
```

- You have the basis now of many compiler optimization opportunities!


```
1 # let counter = something
2 val counter : unit -> int = <fun>
3 # counter ();;
4 - : int = 1
5 # counter ();;
6 - : int = 2
7 # counter ();;
8 - : int = 3
9 #
```

- Can we still use equational reasoning to talk about programs now?

- `f(x) + f(x) == 2 * f(x)`

```
1 # 2 * counter ();;
2 - : int = 8
3 # counter () + counter ();;
4 - : int = 11
```

- Congratulations. You just broke mathematics.

Transition Semantics

`ref` $v \rightarrow \$i$, where $\$i$ is a free location in the state, initialized to v.

! $\$i \rightarrow v$, if state location $\$i$ contains v

$\$i := v \rightarrow ()$, and state location $\$i$ is assigned v.

$();e \rightarrow e$

Note that references are different than pointers: once created, they cannot be moved, only assigned to and read from.

Natural Semantics

$\frac{e \Downarrow v}{\texttt{ref}\ e \ \Downarrow\ \$i}$, where $\$i$ is a free location in the state, initialized to v.

$\frac{e \Downarrow \$i}{!e \Downarrow v}$, if state location $\$i$ contains v.

$\frac{e_1 \Downarrow \$i \quad e_2 \Downarrow v}{e_1 := e_2 \Downarrow ()}$, and location $\$i$ is set to v.

$\frac{e_1 \Downarrow () \quad e_2 \Downarrow v}{e_1; e_2 \Downarrow v}$

Counter, Method 1

```
1  # let ct = ref 0;;
2  val ct : int ref = {contents=0}
3  # let counter () =
4      ct := !ct + 1;
5      !ct;;
6  val counter : unit -> int = <fun>
7  # counter ();;
8  - : int = 1
9  # counter ();;
10 - : int = 2
```


Bad Things for Counter

`ct` is globally defined. Two bad things could occur because of this.

1. What if you already had a global variable `ct` defined?
   - Correct solution: use modules.
2. The Stupid User^{TM} might decide to change `ct` just for fun.
   - Now your counter won't work like it's supposed to. . .
   - Now you can't change the representation without getting tech support calls.
   - Remember the idea of *abstraction*.

Conclusions about State

State is bad because:

- it breaks our ability to use equational reasoning
- users can get to our global variables and change them without permission

State is good because:

- Certain constructs are almost impossible withouth state (e.g., Graphs)
- Our world is a stateful one

Review of Scope

```
1 let x = 10;;
2
3 let foo y = match y with
4  | 0,b -> let c = b * b in
5           let d = c * c in
6              b * c * d
7  | a,b -> map (fun z -> z + a + x) [a;b]
```

- x exists from line 2–7.
- y exists from line 3–7.
- b exists from line 4–6.
- c exists from line 5–6.
- d exists on line 6 only.
- a and b exist on line 7 only.
- z exists on line 7, after the `fun z` until the `)`.

Using local state

```
1 # let counter =
2     let ct = ref 0 in
3     fun () -> ct := !ct + 1; !ct;;
4 val counter : unit -> int = <fun>
5 # counter ();;
6 - : int = 1
7 # counter ();;
8 - : int = 2
```

- This protects `ct`, making it available only to `counter`.

Random Number Generators

```
1  # let mkRandom s =
2      fun () -> s := (!s * 9 + 5) mod 1024; !s;;
3   val mkRandom : int ref -> unit -> int = <fun>
4  # let rnd0 = mkRandom (ref 1);;
5  val rnd0 : unit -> int = <fun>
6  # rnd0 ();;
7  - : int = 14
8  # rnd0 ();;
9  - : int = 131
10 # rnd0 ();;
11 - : int = 160
```

- In this version we pass the reference into the function rather than creating our own.

Function Tuples

```
1  # let (counter, reset) =
2      let ct = ref 0 in
3        (fun () -> ct := !ct + 1; !ct),
4        (fun nv -> ct := nv);;
5  val counter : unit -> int = <fun>
6  val reset : int -> unit = <fun>
7  # counter ();;
8  - : int = 1
9  # reset 5;;    (* This trick brought to you by *)
10 - : unit = ()  (* higher order functions, tuples, *)
11 # counter ();; (* and the principle of orthogonality. *)
12 - : int = 6
```

Passing Counters Around

```
# let enumerate lst (ctfun, rsfun) =
    rsfun 0;
    List.map (fun x -> (ctfun (), x)) lst;;
val enumerate : 'a list ->
  (unit -> 'b) * (int -> 'c) -> ('b * 'a) list = <fu
# enumerate ["hello";"there";"class"]
          (counter, reset);;
- : (int * string) list = [1, "hello"; 2, "there";
                           3, "class"]
#
```

- We can give the counter to another function.
- This is not good. Why not?

Activity

1. Supposing you wanted a counter that did *not* use references, how would you go about writing it?
2. The random number function generator does not have a way to reset the state. We would also like to be able to ask "what was the last random number generated" without changing the seed. Write a (group of) functions to do this.
3. Suppose we want a more generic way to represent counters—in fact, suppose you want *several* counters in your program, each with an init and a reset. You could just repeat the code several times, but there are serious flaws to that approach. What are they? How might you go about fixing them?

Answer to 2

```
# let mkRandom init =
    let seed = ref init in
   (( fun () -> seed := !seed * 4; !seed),
    ( fun ns -> seed := ns),
    ( fun () -> !seed) );;
        val mkRandom : int -> (unit -> int) * (int ->
# let (rnd, reset, last) = mkRandom 4;;
val rnd : unit -> int = <fun>
val reset : int -> unit = <fun>
val last : unit -> int = <fun>
```


Preliminaries

- We will use the following functions during our discussion....

```
let pi1 (x,y) = x
let pi2 (x,y) = y
let report (x,y) = print_string "Point: ";
                   print_int x;
                   print_string ",";
                   print_int y;
                   print_newline ()
let movept (x,y) (dx,dy) = (x+dx,y+dy)
```

Here is an example of a point using local state.

```
let mktPoint myloc =
  let myloc = ref myloc in
     ( myloc,
       (fun () -> pi1 !myloc),
       (fun () -> pi2 !myloc),
       (fun () -> report !myloc),
       (fun dl -> myloc := movept !myloc dl) )
```

- This defines a tuple of functions that share a common state.
- It is cumbersome to use.

```
let (lref,getx,gety,show,move) = mktPoint (2,4);;
```



```
type point = {
  loc : (int * int) ref;  getx : unit -> int;
  gety : unit -> int;  draw : unit -> unit;
  move : int * int -> unit;
}
let mkrPoint newloc =
   let myloc = ref newloc in
   { loc = myloc;
     getx = (fun () -> pi1 !myloc);
     gety = (fun () -> pi2 !myloc);
     draw = (fun () -> report !myloc);
     move = (fun dl -> myloc := movept !myloc dl)}
```


What is an object?

- Data and functions are grouped together.
- Functions have their own local state.
- Objects can send and receive *messages*.
- Objects can refer to *themselves*.

This has a profound effect on the way programs are written. Remember the basic premise of this course: how you think about data has a great impact on the way a program is written.

- How is the `mkrPoint` example like an object?
- How is the `mkrPoint` example not like an object?

By the way, this lecture is really about recursion.

```
let mkPoint newloc =
  let rec this =
  { loc = ref newloc;
    getx = (fun () -> pi1 !(this.loc));
    gety = (fun () -> pi2 !(this.loc));
    draw = (fun () -> report !(this.loc));
    move = (fun dl ->
              this.loc := movept !(this.loc) dl)}
 in this;;
```

We can store “this” explicitly in the record if we want.

- The record `point` contains references to the fields. If you copy a `point`, the data does **not** get copied!

```
# let p1 = mkPoint (4,7);;
val p1 : point = {loc={contents=4, 7}; ...}
# let p2 = mkPoint (6,2);;
val p2 : point = {loc={contents=6, 2}; ...}
# let p3 = p1;;
val p3 : point = {loc={contents=4, 7}; ...}
# p1.move (5,5);;
- : unit = ()
# p3;;
- : point = {loc={contents=9, 12}; ...}
```


We used a record to implement a type for points.
Advantages:

- Every method had its own name and type.
- Simple syntax for manipulating the object.
- It's fast: we know at compile time which method is being called.

Disadvantages:

- Inheritance is very difficult with this model.
- Adding a new message type means updating *everything*.

Last time we said that an object is a kind of data that can *receives messages* from the program or other objects.

- Q: How do we normally represent messages?
- A: With strings!

Let a point object be a function which takes a string and returns an appropriate function matching that string.

- Question: Suppose p is our point object. What will be its type?


```
let mkPoint x y =
  let x = ref x in
  let y = ref y in
  fun st ->
    match st with
    | "getx" -> (fun _ -> !x)
    | "gety" -> (fun _ -> !y)
    | "movx" -> (fun nx -> x := !x + nx; nx)
    | "movy" -> (fun ny -> y := !y + ny; ny)
    |  _ -> raise (Failure "Unknown message.")
```

All methods now have to have type `int -> int`.

Subclassing

- Warmup exercise: How would we add a `report` method?
- Another one: How would we add `this` support?

Let's say we want a `fastpoint`, which moves twice as fast as the original point. What does it mean for `fastpoint` to be a *subclass* of `point`?

- `fastpoint` should respond to the same messages.
  - It may override some of them.
  - It may add its own.
  - It may **not** remove any methods.
- The `fastpoint` object will need access to some of the data in `point`.

Implementing

- Two entities involved: the superclass (`point`) and the subclass (`fastpoint`).
- `fastpoint` needs to create an instance of `point`.
- `point` construction needs to return the "public" data to `fastpoint`.
- `fastpoint` returns a dispatcher:
  - if the `fastpoint` dispatcher can handle a message, it does.
  - Otherwise, it *sends the message* to `point`.

Code: `point`

```
1  let mkSuperPoint x y =
2    let x = ref x in
3    let y = ref y in
4    ((x,y),  (* This part returns the local state *)
5    fun st ->
6      match st with
7      | "getx" -> (fun _ -> !x)
8      | "gety" -> (fun _ -> !y)
9      | "movx" -> (fun nx -> x := !x + nx; nx)
10     | "movy" -> (fun ny -> y := !y + ny; ny)
11     |  _ -> raise (Failure "Unknown message."));;
12 val mkSuperPoint : int -> int ->
13    (int ref * int ref) * (string -> int -> int) = <fu
```


Code: `fastpoint`

```
1  let mkFastpoint x y =
2    let ((x,y),super) = mkSuperPoint x y in
3    fun st ->
4      match st with
5      | "movx" -> (fun nx -> x := !x + 2 * nx; nx)
6      | "movy" -> (fun ny -> y := !y + 2 * ny; ny)
7      |  _ -> super st;;
```

- This technique is flexible; we can add methods very easily.
- But it's also slow. Imagine if we had a chain of 20 classes....

C++

- Methods and variables are kept in a table: a fixed location.
- "`this`" is an implicit argument, allowing only one copy of the function to be needed.
- Virtual methods are kept in a *vtable*, which counts as local data.

Local data for `point` or `fastpoint`:

`x`	*value of x*
`y`	*value of y*
`vtable`	*pointer to vtable*

Vtable for `point`:

`movx`	*pointer to point.movx*
`movy`	*pointer to point.movy*

(`fastpoint` vtable is similar.) `getx`, etc. is static.

Discussion

- Other languages (i.e., smalltalk) use a technique very similar to this one.
- Java uses the "every object is of type `Object`" technique.
- A strong type system makes it somewhat cumbersome to simulate objects. You either have to:
  - define a new type to encompass all objects, or
  - force all methods to have the same type.
- Important concept: *polymorphism* — when functions can operate on multiple types. (This is different than *overloading* — when multiple functions exist with the same name, but different inputs.)

Polymorphism Example

```
# let p1,p2,p3,p4 = mkPoint 2 3, mkPoint 3 2,
            mkFastpoint 5 3, mkFastpoint 3 9;;
# List.map (fun pt -> pt "report" 0)
           [p1; p2; p3; p4];;
Point: 2,3  points
Point: 3,2
Point: 5,3  fastpoints
Point: 3,9
```

The function passed to `map` will use both `point` and `fastpoint` types.
You have seen polymorphism in the course before.

Conclusions

- Objects have a lot of flexibility, and allow us to create useful abstractions.
- They can be implemented using functions.
- These are useful enough in practice, and difficult enough to implement, that most modern languages now include them, including OCaml. (That's where the O comes from.)